# Harmonic Scheduling of Linear Recurrences in Digital Filter Design *

Haigeng Wang wang@ics.uci.edu
Nikil Dutt dutt@ics.uci.edu
Alexandru Nicolau nicolau@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Technical Report Number 92-13

February 14, 1992

**Abstract**

Linear difference equations involving recurrences are fundamental equations that describe many important signal processing applications. For many high sample rate digital filter applications, we need to effectively parallelize the linear difference equations used to describe digital filters – a difficult task due to the recurrences inherent in the data dependences. We present a novel approach, *Harmonic Scheduling*, that exploits parallelism in these recurrences beyond *loop-carried dependencies*, and which generates optimal schedules for parallel evaluation of linear difference equations with resource constraints. This approach also enables us to derive a parallel schedule with minimum control overhead, given an execution time with resource constraints. We also present a Harmonic Scheduling algorithm that generates optimal schedules for digital filters described by second-order difference equations with resource constraints.

*This work was supported in part by NSF grants CCR8704367, MIP9009239 and ONR grant N0001486K0215.

# 1 Introduction

Digital filter designs are typically described using linear difference equations that express the filter's current response $y(nT)$ as a function of past responses:

$$\begin{aligned} y(nT) &= a_0x(nT) + a_1x[(n-1)T] + a_2x[(n-2)T] + \cdots + a_{k-1}x[(n-k+1)T] + \\ &\quad + b_1y[(n-1)T] + b_2y[(n-2)T] + \cdots + b_my[(n-m)T]. \end{aligned} \tag{1}$$

Here the filter response at time $nT$, where $T$ is the sample period and $n$ is a running index, is the sum of product terms of $m$ past responses, $y[(n-1)T], \ldots, y[(n-m)T]$, where $m$ is the *order* of the difference equation. The pattern of data dependences exhibited by (1) involves recurrences, since $t$ is a linear combination of the outputs at time $t-1, t-2, \ldots, t-m$. This recurrent dependence pattern is often referred to as a *loop-carried dependence*(LCD). The difference equation (1) is a special case of *band linear recurrences.* In many high sample rate applications, it is crucial to parallelize the classes of signal processes described with the linear difference equation, in order to maximize throughput given a limited number of resources (e.g., functional units or chip area). In the high-level synthesis of high performance signal processors, we need effective algorithms for: (1) minimizing resources given a throughput constraint for a digital filter, or (2) maximize throughput (i.e., generate optimal schedules), given a resource constraint for the filter. Both of these kinds of algorithms require effective techniques for parallelizing difference equations.

However, difference equations are difficult to parallelize due to the LCD's in the recurrences, particularly when resource constraints are imposed. Previous research in scheduling for high level synthesis has exploited parallelism for signal processing in the forms of pipelining (i.e., overlapping the processing of a sample input with subsequent inputs), and concurrency (i.e., simultaneously computing operations that may be executed in parallel in processing one sample sequence while preserving data dependences). Although these scheduling techniques produce good results, we can further exploit parallelism by taking advantage of the recurrent patterns described by LCDs. This is important for a difference equation (1) that exhibits LCDs, since there is not much parallelism within the LCD's: i.e., the speedup of throughput with current parallelization techniques is bounded by a small constant regardless of how many functional units are used.

In this paper, we present an approach, *Harmonic Scheduling*, that devises optimal schedules for parallel evaluation of linear difference equations with resource constraints in the context of high level synthesis of digital filters. Harmonic Scheduling[11, 12] generates optimal parallel schedules for evaluating band linear recurrences. The paper is organized as follows. Section 2 briefly reviews the previous work in scheduling for high level synthesis and parallel linear recurrence evaluation. Section 3 lists the assumptions and definitions. Section 4 highlights the idea of using Harmonic Scheduling to exploit recurrences. Section 5 gives an algorithm for generating optimal schedules for 2nd-order difference equations, and illustrates the algorithm with an example. Section 6 summarizes the paper.

## 2 Previous Work

Related work has been done in two areas: scheduling algorithms for high level synthesis and parallel computing. Scheduling algorithms for high level synthesis have been exploiting three forms of parallelism: concurrency, functional pipelining and loop pipelining. Concurrency[2] explores instructions that may be executed simultaneously. Functional pipelining techniques overlap instructions along the data paths [2, 13]. Loop pipelining techniques transform a sequential loop into a loop with parallelism across multiple iterations extracted while preserving the program's semantics. Since scheduling with resource constraints in general is NP-complete, heuristic-based loop pipelining techniques [4, 16, 6] have been developed to compact loops with given resource constraints. Percolation-based loop pipelining techniques [15] first compact a loop into its optimal parallel counterpart and then apply resource constraints on the parallel version. All of the techniques above preserve the LCD's, and hence do not exploit recurrence structures to gain further parallelism. Techniques that extract parallelism beyond LCD's have just started to receive attention in the high level synthesis and digital signal processing communities. A few techniques based on tree height reduction [5, 10, 9] have been proposed to reduce, using extra operations, the length of critical paths consisting of a sequence of additions.

In the field of parallel computing, parallel evaluation of linear recurrences has been studied for quite some time. However, the underlying model differs from high-level synthesis, since parallel computing typically deals with a number of identical, multi-function processors, while high-level synthesis deals with a variety of functional units that differ in several attributes (e.g., functionality, cost, speed). We briefly review some results from parallel computing. Kogge and Stone[8] described a technique called *recursive doubling* for computing the first-order linear recurrence system with unlimited resources. Chen and Kuck developed an algorithm [1] for computing $m$-th order band linear recurrence with $p$ processors that achieves a time bound of $(n/p)(2m^2 + 3m) + O(n^2 log(p/m)$. Hyafil and Kung[7] established a pair of lower and upper bounds of $3N/(p+1/2)$ and $3N/(p+1/2)+O(\log p)$ time steps respectively with $p$ processors for parallel evaluation of the Horner expression, which is equivalent to evaluating the last equation in a first-order band linear recurrence, i.e., evaluating $x_N$ only without having to compute $x_1, ..., x_{N-1}$. Gajski[3] lowered the time bound of Chen further to $(2m^2+3m)N/(p+m+1/2)$ for $p \geq m+1$ and $N > p^2$ for computing the band linear recurrences with $p$ MIMD processors. Recently, Wang and Nicolau[12] proposed a novel approach, Harmonic Scheduling, which generates optimal schedules for linear recurrences that achieve an execution time of

$$\frac{(2m^2 + 3m)N}{p + \frac{(m(m+1)(2m+1))}{2(2+\lfloor \log m \rfloor)}}, \text{ for } m \geq 1, p \geq \frac{(m^3 + 3/2m^2 - 1/2m)}{(2 + \lfloor \log m \rfloor)}.$$

When $p < \frac{(m^3+3/2m^2-1/2m)}{(2+\lfloor \log m \rfloor)}$, their schedules also give better execution time than the previously published fastest schedules. They proved that this is the strict time lower bound for $m = 1, 2, 3$. In addition to the time bounds, their method also facilitates deriving schedules with best program-space efficiency (so that a schedule can better fit into caches), given an execution time.

# 3 Assumptions, Terminology and Definitions

In order to highlight the technique for exploiting recurrences, we make some simplifying assumptions about the target architecture and the functional units used. The technique we describe is still applicable if these assumptions are relaxed to model more realistic architectures with a range of functional units. We assume that the target architecture for the DSP consists of a set of functional units (*FUs*) that can perform a logic operation, an addition or a multiplication in one time step (i.e., unit time). We assume that each functional unit has a dedicated control unit (with a register-file or memory for microinstructions) that controls the functional unit in each time step. We also ignore the effect of the number of intermediate registers (storage units) required to store the results of redundant operations. Further, we assume that we are given an initial allocation of functional units (resources). Our objective is to generate an optimal schedule (i.e., minimal number of time steps) for this allocation that exploits LCD recurrences, followed by a minimization of the control overhead for each functional unit (i.e., the size of the microprogram for each functional unit).

We now define the terms frequently used in this paper. A computation $A$ can be performed using a sequential schedule $A_{seq}$ on a design with a single functional unit, or using a parallel schedule $A_{par}$ on a design with $p$ functional units. [1]. We denote the time to run a schedule $A_{par}$ on a design with $p$ functional units as $T_p(A_{par})$ (or $T_p$ or $T$ when unambiguous). We refer to $T_p$ and $T$ interchangeably as execution time, time steps or steps. The time to compute $A$ sequentially (on a single functional unit) is denoted by $T_1(A_{seq})$. The *speedup* of a design with $p$ functional units over a design with a single functional unit for computation $A$, is denoted $S_p(A) = T_1(A_{seq})/T_p(A_{par})$, or simply $S_p = T_1/T_p$ when unambiguous. The *efficiency* of this computation is $E_p = S_p/p$, which can be interpreted as actual speedup divided by the maximum possible speedup using $p$ functional units. Let $O_p$ be the number of operations executed in some computation using $p$ functional units. We define *operation redundancy* to be $R_p = O_p/O_1(\geq 1)$, where $O_1 = T_1$. Finally, we define *utilization* as $U_p = O_p/pT_p \leq 1$, where $pT_p$ is the maximum number of operations that $p$ functional units can perform in $T_p$ steps. The *final values* of a linear recurrence are the values computed by the definition of the recurrence, i.e., the values assigned to the left-hand side of the statements in the loop body shown before. The *final operations* of a linear recurrence are operations that compute the final values in the sequential computation. The *redundant operations* are operations that compute auxiliary values introduced by the parallel schedules in an effort to speed up the computation of the final values. The *redundant values* refer to the auxiliary values computed by the redundant operations.

We introduce some terms for the matrix representation of linear recurrences illustrated using first-order linear recurrences. The sequential evaluation of $N$ first-order linear recurrences can be expressed

[1] We distinguish between our *algorithm* and *schedule* – our algorithm produces a schedule (for parallel evaluation of the given linear recurrence) which, when run on a set of functional units, actually evaluates the linear recurrence in parallel.

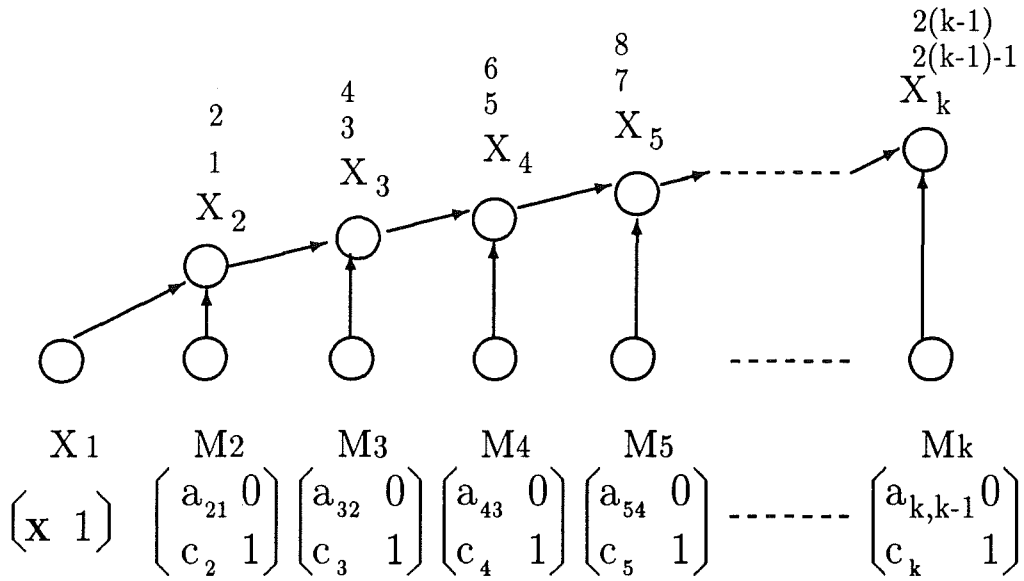


Figure 1: A dependence tree for the sequential computation of a 1st-order linear recurrence

as a chain of matrix multiplications:

$$[x_k\ 1] = [x_1\ 1] \begin{bmatrix} a_{21} & 0 \\ c_2 & 1 \end{bmatrix} \begin{bmatrix} a_{32} & 0 \\ c_3 & 1 \end{bmatrix} \begin{bmatrix} a_{43} & 0 \\ c_4 & 1 \end{bmatrix} \cdots \begin{bmatrix} a_{k,k-1} & 0 \\ c_k & 1 \end{bmatrix}$$

for $1 \leq k \leq N$. This is referred to as the *matrix decomposition* of the evaluation. The sequential matrix chain multiplication above can be represented as a dependence tree as shown in Figure 1.

The leaves of the computation tree, $X_1, M_2, ..., M_k$, represent the matrices, called the *operand nodes*. The nodes on the top fringe of the tree, $X_1, ..., X_k$, represent the results of each recurrence in the system packaged in matrix form, called the *result nodes* or *final nodes*. A result node represents the result of the multiplication of the two matrices below it, i.e., $X_i = X_{i-1}M_i$ for $2 \leq i \leq k$. In a matrix multiply only a sequence of two arithmetic operations, a multiply followed by an add, are necessarily done. The column of numbers above a result node represents the time steps at which the single functional unit is allocated to execute the operations in that node. The number of operations in a sequential evaluation of $N$ $m$-th order banded linear recurrences is $2mN$. Clearly, the computation above cannot be significantly parallelized without breaking the dependences and introducing redundant operations, because each result node $X_i, 2 \leq i \leq k$, depends on the previously computed result node $X_{i-1}$.

By introducing redundant operations, a parallel evaluation of a system of eight 1st-order linear recurrence can be represented as the tree shown in Figure 2. In addition to the result nodes and operand nodes, a parallel computation tree uses the *redundant nodes*, $R_1, ..., R_5$ in Figure 2. A redundant node represents the matrix resulting from the multiplication of two nodes with dependence arcs reaching it, for example, $R_1 = M_3M_4$ and $R_4 = R_2M_7$. For first-order linear recurrences, a redundant matrix

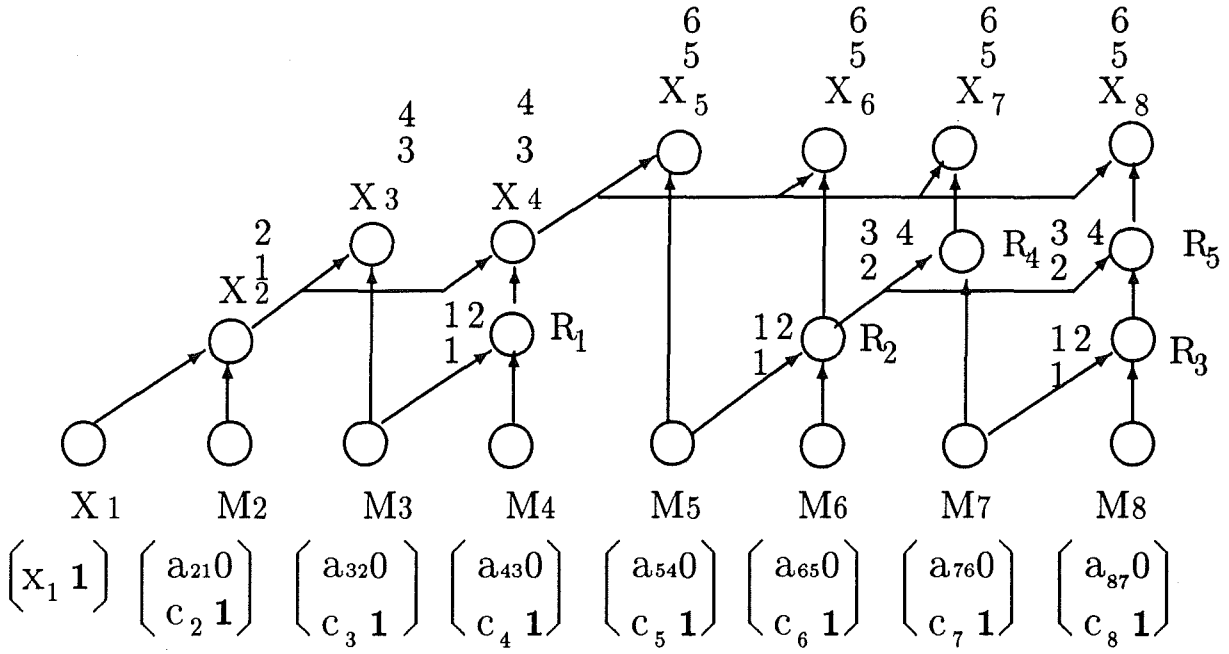


Figure 2: A dependence tree for parallel computation of a 1st-order linear recurrence

is composed of three arithmetic operations, which can be done in two parallel chains of arithmetic operations, one consisting of a multiply followed by an add, called the *long chain*, and the other consisting of one add, called the *short chain*. As for the result nodes, the columns of numbers associated with the redundant nodes represent the time steps at which a functional unit is allocated to execute the operations in those nodes. The parallel schedule in Figure 2 computes the given recurrence in six steps on seven functional units. Although the tree height is lowered with parallelism resulting from redundant matrix multiplications, the functional unit utilization is bad: $U_7 = 27/42$.

A matrix multiplication in a schedule is called a *component*. All components in a schedule can be classified into different types, based on the number of arithmetic operations in the component. Three types of components are needed in parallel schedules for first-order linear recurrences as shown in Figure 3 (in general, $(m+1)$ types are needed for $m$th order linear recurrences for $m > 1$). The first type of component is composed of an operand node with a dependent result node (or equivalently, a result node with a reaching operand node). For example, $X_2$ with $M_2$ and $X_3$ with $M_3$ make two components of the first type respectively. The second type of component is composed of two result nodes that depend on a redundant node and two operand nodes, for example, nodes $X_4, X_5, R_1, M_4$ and $M_5$ make a component of the second type. The third type of component is composed of a result node, a redundant node and an operand node chained by dependences, for example, node $X_6, R_2$ and $M_6$ make such a component. These three types of components are sufficient for constructing optimal parallel schedules for a first-order linear recurrence.

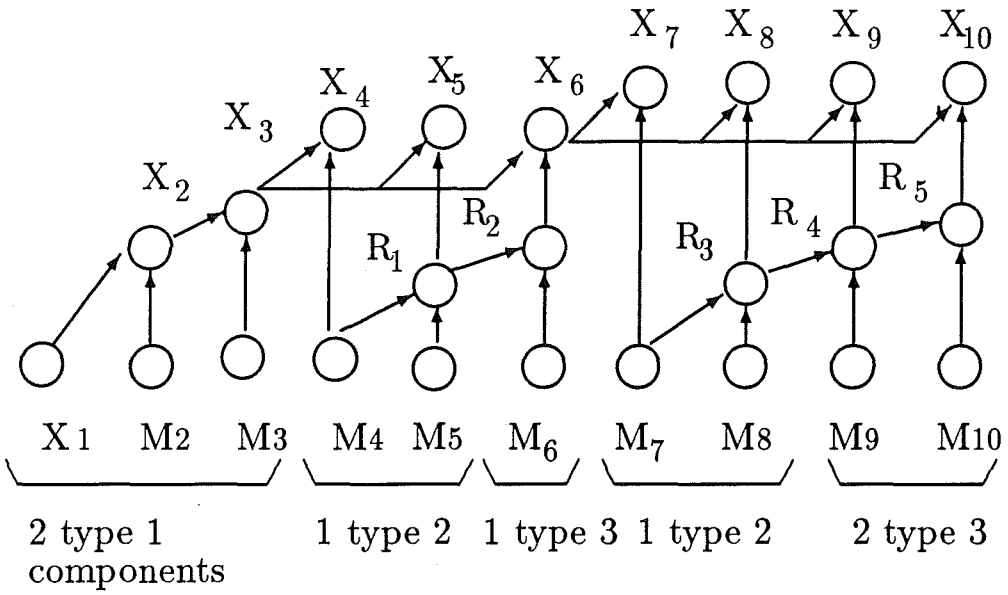


Figure 3: Three types of components in a parallel schedule for first-order banded linear recurrence.

## 4 Harmonic Scheduling for Exploiting Band Linear Recurrences

The basic objective of Harmonic Scheduling [11, 12] is to simultaneously minimize the number of redundant operations and maximize the utilization of the allocated functional units; we call these tasks the *dual requirements* of Harmonic Scheduling. We know that the number of final operations in any schedule equals at least the number of operations in the sequential schedule, which cannot be reduced due to the definition of the required outputs. The only mechanism for significantly speeding up the computation is to use multiple functional units to compute redundant values ahead of the final value computation; this shortens the critical path in the computation of some final values, and makes available multiple final values in the fewest possible parallel steps by using previously computed (final and redundant) values.

In order to compute as many final values as possible, given a fixed number of functional units, a schedule should do as few redundant operations as possible. However, one cannot reduce the number of redundant operations arbitrarily, say, to zero, since this will sequentialize the computation of final results(thus making a parallel schedule degenerate to a sequential schedule in the limit), and will reduce the speedup to a minimum. Intuitively, the fastest parallel schedules for a fixed number of functional units should and would satisfy the dual requirements. Such a state, in which the dual requirements are satisfied, can be seen as having achieved "harmony" among the conflicting goals, hence the name "Harmonic Scheduling".

A parallel schedule can be divided into a number of *periods* that have the same organization. Such a period is constructed with a combination of all types of components satisfying the dual requirements. The optimality of the entire parallel schedule may be proved based on the optimality of each period. The numbers of all types of components to be used in constructing an optimal period are simply the

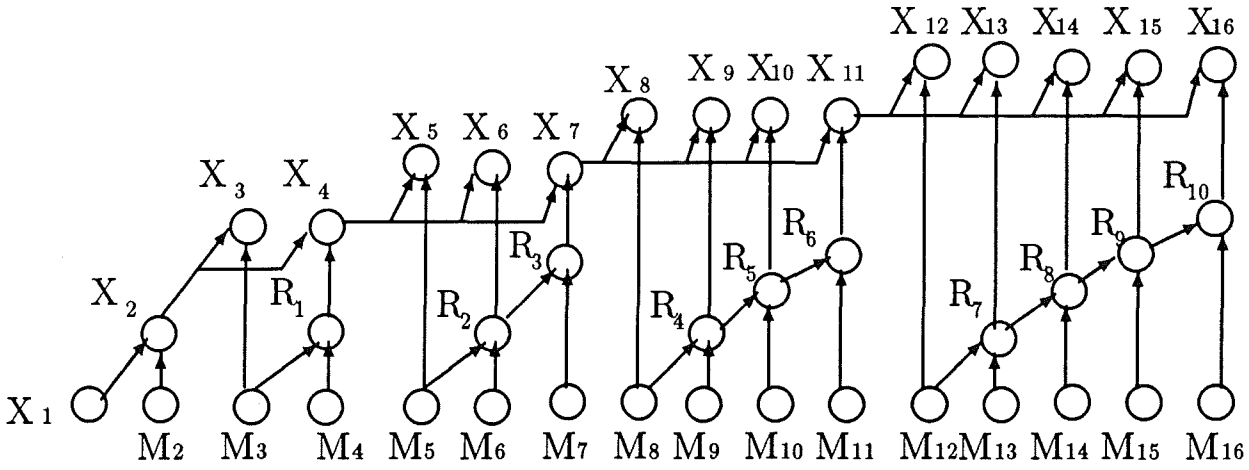

Figure 4: The computation tree for $p = 6$.

integer solutions for the linear inequalities describing the relationship of the execution time to the pattern of the schedules. With this information, the computation tree for the period is constructed, and functional unit slots are assigned to all arithmetic operations in the tree. The complete schedule is finally obtained by constructing the computation tree at the operation level.

We illustrate this procedure with an example. For $p = 6$ functional units, an optimal schedule needs one component of the first type, four of the second type and six of the third type. Its computation tree in terms of matrix multiplication is constructed as shown in Figure 4. To facilitate functional unit assignment, that tree is further expanded into an equivalent tree in terms of arithmetic operations as shown in Figure 5.

In Figure 5, the boxes represent the arithmetic operations in the computation tree for a period. The numbers inside the boxes represent the time steps at which functional unit slots compute the operations. Each column of boxes in the two top rows corresponds to a final node. Each column of boxes in the three bottom rows corresponds to a redundant node. The numbers associated with redundant operation boxes are the indices for a temporary array $t$ that stores the redundant values.

In each time step, we have $p = 6$ processor slots. In step one, we allocate one processor slot to the first operation on the critical path, and five slots to the redundant trees, four of which are allocated to the four long chains and the last one to the first short chain. We thus can see all the operation boxes marked with "1". Step two is done similarly to step one: the first slot is allocated the operation on the critical path, and the other five allocated to the redundant tree, four of which are allocated to the second operations on all the long chains and the last slot to the second long chain. In step five, we allocate the first slot to the operation on the critical path, and the next four to the operations in the third and fourth redundant trees. Then there are no available operations in the redundant trees for the last slot in step five; thus we fill the slot with an available final operation that is not on the critical path. The allocation proceeds until in step ten all the remaining final operations are allocated—our 60 processor slots are precisely filled with 60 operations in the computation tree for a period.

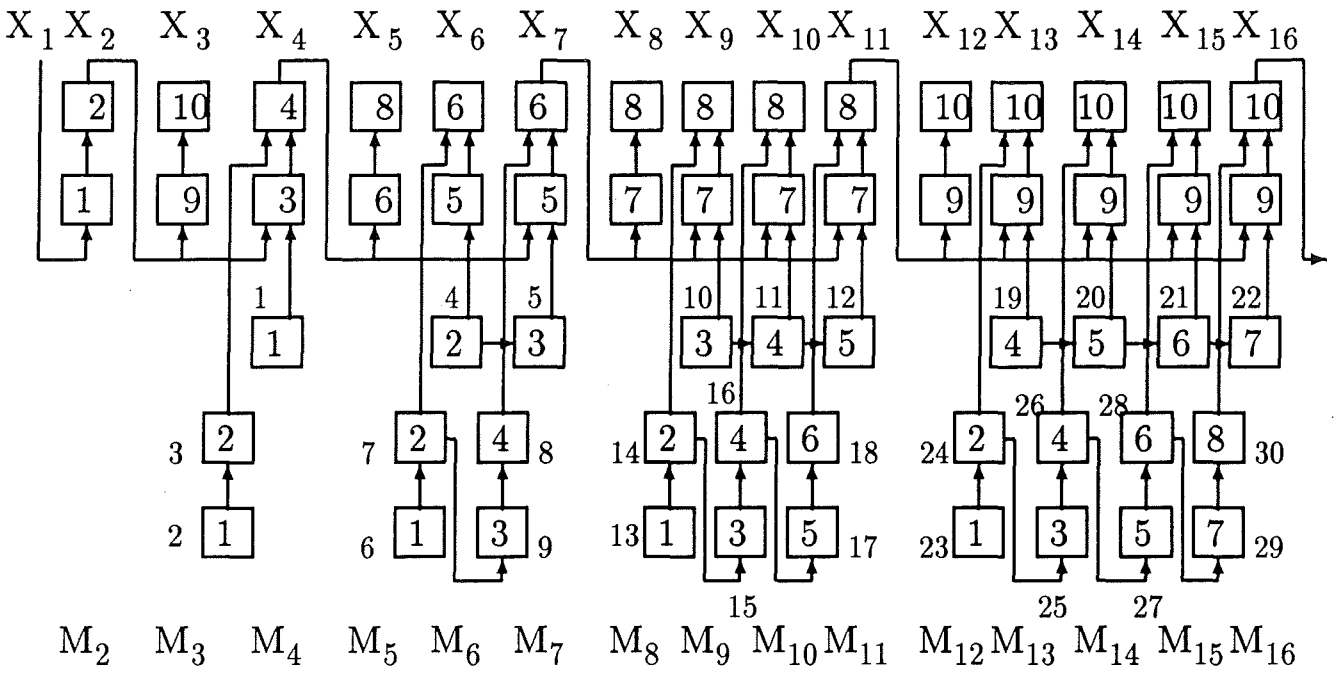


Figure 5: The computation tree with allocated functional unit slots for $p = 6$.

In the final schedule, array $a[N][N]$ holds the coefficients, $c[N]$ holds the constant terms, $x[N]$ holds the results and $t[\eta]$ holds the results of the redundant operations of the first-order banded linear recurrence, where $\pi$ is the period length in terms of number of outputs, $\eta$ is the number of redundant operations in a period, $\eta = 30$ for $p = 6$. Obviously, a careful use of memory can reduce memory size to an order of $N$. We chose to neglect the memory efficiency in the following schedule in favor of readability.

**for** $i = 0, ..., N/\pi - 1$ **do** *each step in parallel with p functional units*

1. $x[\pi k+2] = a[\pi k+2][\pi k+1] * x[\pi k+1];$
   $t[2] = c[\pi k+3] * a[\pi k+4][\pi k+3];$
   $t[13] = c[\pi k+8] * a[\pi k+9][\pi k+8];$
2. $x[\pi k+2] = x[\pi k+2] + c[\pi k+2];$
   $t[7] = t[6] + c[\pi k+6];$
   $t[24] = t[23] + c[\pi k+13];$
3. $x[\pi k+4] = t[1] * x[\pi k+2];$
   $t[15] = t[14] * a[\pi k+10][\pi k+9];$
   $t[5] = t[4] * a[\pi k+7][\pi k+6];$
4. $x[\pi k+4] = x[\pi k+4] + t[3];$
   $t[16] = t[15] + c[\pi k+10];$
   $t[11] = t[10] * a[\pi k+10][\pi k+9];$
5. $x[\pi k+6] = t[4] * x[\pi k+4];$
   $t[17] = t[16] * a[\pi k+11][\pi k+10];$
   $t[12] = t[11] * a[\pi k+11][\pi k+10];$
6. $x[\pi k+6] = x[\pi k+6] + t[7];$
   $x[\pi k+4] = x[\pi k+3] * a[\pi k+5][\pi k+4];$
   $t[28] = t[27] + c[\pi k+15];$
7. $x[\pi k+8] = x[\pi k+7] * a[\pi k+8][\pi k+7];$
   $x[\pi k+10] = x[\pi k+7] * t[11];$

   $t[1] = a[\pi k+3][\pi k+2] * a[\pi k+4][\pi k+3];$
   $t[6] = c[\pi k+5] * a[\pi k+6][\pi k+5];$
   $t[23] = c[\pi k+3] * a[\pi k+4][\pi k+3];$
   $t[3] = t[2] + c[\pi k+4];$
   $t[14] = t[13] + c[\pi k+9];$
   $t[4] = a[\pi k+5][\pi k+4] + a[\pi k+6][\pi k+5];$
   $t[8] = t[7] * a[\pi k+7][\pi k+6];$
   $t[25] = t[24] * a[\pi k+14][\pi k+13];$
   $t[10] = a[\pi k+8][\pi k+7] * a[\pi k+9][\pi k+8];$
   $t[9] = t[8] + c[\pi k+7];$
   $t[26] = t[25] + c[\pi k+14];$
   $t[19] = a[\pi k+12][\pi k+11] * a[\pi k+13][\pi k+12];$
   $x[\pi k+7] = t[5] * x[\pi k+4];$
   $t[27] = t[26] * a[\pi k+15][\pi k+14];$
   $t[20] = t[19] * a[\pi k+14][\pi k+13];$
   $x[\pi k+7] = x[\pi k+7] + t[9];$
   $t[18] = t[17] + c[\pi k+11];$
   $t[21] = t[20] * a[\pi k+15][\pi k+14];$
   $x[\pi k+9] = x[\pi k+7] * t[10];$
   $x[\pi k+11] = x[\pi k+7] * t[12];$

$t[29] = t[28] * a[\pi k + 16][\pi k + 15];$
8. $x[\pi k + 8] = x[\pi k + 8] + c[\pi k + 8];$
$x[\pi k + 10] = x[\pi k + 10] + t[16];$
$x[\pi k + 5] = x[\pi k + 5] + c[\pi k + 5];$
9. $x[\pi k + 12] = x[\pi k + 10] * a[\pi k + 12][\pi k + 11];$
$x[\pi k + 14] = x[\pi k + 11] * t[20];$
$x[\pi k + 16] = x[\pi k + 11] * t[22];$
10. $x[\pi k + 12] = x[\pi k + 12] + c[\pi k + 12];$
$x[\pi k + 14] = t[26] + x[\pi k + 14];$
$x[\pi k + 16] = t[30] * x[\pi k + 16];$
**end for**

$t[22] = t[21] * a[\pi k + 16][\pi k + 15];$
$x[\pi k + 9] = x[\pi k + 9] + t[14];$
$x[\pi k + 11] = x[\pi k + 11] + t[18];$
$t[30] = t[29] + c[\pi k + 16];$
$x[\pi k + 13] = x[\pi k + 11] * t[19];$
$x[\pi k + 15] = x[\pi k + 11] * t[21];$
$x[\pi k + 2] = x[\pi k + 1] * a[\pi k + 3][\pi k + 2];$
$x[\pi k + 13] = t[24] + x[\pi k + 13];$
$x[\pi k + 15] = t[28] + x[\pi k + 15];$
$x[\pi k + 3] = x[\pi k + 3] + c[\pi k + 3];$

# 5 A Scheduling Algorithm for Second-Order Digital Filters

In this section, we derive parallel schedules with resource constraints for second-order recursive filter sections using Harmonic Scheduling. In filter design, the sensitivity analysis indicates that a less-sensitive filter structure may be obtained by breaking up the transfer function into lower-order sections and connecting these sections in parallel or in cascade. Although high-order blocks may be attractive in some applications, the second-order section is a good building block to use in parallel or cascade structures[14]. Needless to say, the Harmonic Scheduling method applies to difference equation in general (1).

The difference equation for the second order section is obtained from (1) by setting all the coefficients to zero except for $a_0, b_1$ and $b_2$, as follows. For simplicity of presentation, we also $a_0 = 1$, thus we have a recursive second-order all pole section with complex conjugate poles [17]. With little modification, the same schedule (for the simple second-order section) can be applied to general second-order sections.

$$y(n) \quad = \quad a_0 x(n) + b_1 y(n-1) + b_2 y(n-2). \tag{2}$$

This is a second-order linear recurrence with constant coefficients, which if parallelized with LCD-preserving techniques would give no further speedup beyond two functional units. Further, $y(n)$ can be expanded into the following form in terms of matrix multiplications

$$[y_n \; y_{n-1} \; 1] = [y_1 \; y_0 \; 1] \begin{bmatrix} b_1 & 1 & 0 \\ b_2 & 0 & 0 \\ x_2 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 & 1 & 0 \\ b_2 & 0 & 0 \\ x_3 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_1 & 1 & 0 \\ b_2 & 0 & 0 \\ x_4 & 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} b_1 & 1 & 0 \\ b_2 & 0 & 0 \\ x_n & 0 & 1 \end{bmatrix}, \tag{3}$$

where $y_i, x_i$ stand for $y(i), x(i)$ respectively. For any redundant matrix multiplication in the chain, the results in the first two rows of the product can be precomputed since they remain the same through all periods of a schedule. Based on the number of arithmetic operations in a matrix multiplication above, three types of matrix multiplication will be involved in our parallel schedules, the first type

using 4 arithmetic operations, the second type using 10 and the third type using 8 respectively. Next, we go through the Harmonic Scheduling phases for difference equation (2).

In the first phase, we solve some inequalities for the numbers of all types of components to be used in our parallel schedule. The Harmonic Scheduling establishes a system of inequalities as follows and asks for integer solutions.

$$\begin{aligned} \frac{4w_0 + 10w_1 + 8w_2}{w_0 + 2w_1 + w_2} &= \frac{T_p}{N}p, \\ 2w_0 + 3w_1 &\leq \frac{4w_0 + 10w_1 + 8w_2}{p}, \\ w_0 &\geq 0, \\ w_1 &> 0, \\ w_2 &\geq 0. \end{aligned} \tag{4}$$

In the inequalities, $w_0, w_1, w_2$ are numbers of the first, second, and third type components respectively, and $p$ is the number of functional units. The first equation relates the pattern of a period of a parallel schedule to its execution time. It essentially says that, in order to achieve an execution time $T_p$ for $p$ functional units, a parallel schedule must produce $w_0 + 2w_1 + w_2$ final results (i.e., sample outputs) using every $4w_0 + 10w_1 + 8w_2$ operations. The other inequalities serve as necessary conditions for the feasibility of solutions. The second inequality says that a solution would give an *infeasible* schedule (i.e., the schedule cannot achieve the execution time $T_p$), if the length of the critical path in a period (described by the left-hand side) were greater than the number of steps required by the number of operations. The other three simple inequalities constrain the solution elements to be non-negative. Specifically, a parallel schedule must have some second type components.

We use execution time $T_p = (8p - 4)N/(p(p + 1))$ in inequalities (4) , which gives a speedup of $(p + 1)/(2 - 1/p) > p/2$ over sequential schedules – the best time so far. Although it is not the strict time lower bound, the strict time lower bound can be achieved by lowering $T_p$ in inequalities (4). A set of integer solutions to inequalities (4) parameterized for $p$ is given below. Not all integer solutions given by those expressions can be made into feasible schedules. But for any $p > 2$, there exists $t_0$ such that a feasible schedule can be built out of the solutions. A detailed discussion is given in [12].

$$\begin{aligned} w_0 &= 0 \\ w_1 &= 2t_0 \\ w_2 &= (p - 3)t_0 \end{aligned} \tag{5}$$

The following algorithm constructs a feasible schedule using the solutions above.

**Input:** $p$ functional units for constructing a schedule for parallel evaluation of the second-order difference equation (2), with $w_0$ components of the first type, $w_1$ of the second type and $w_2$ of the third type.

**Output:** a parallel schedule.

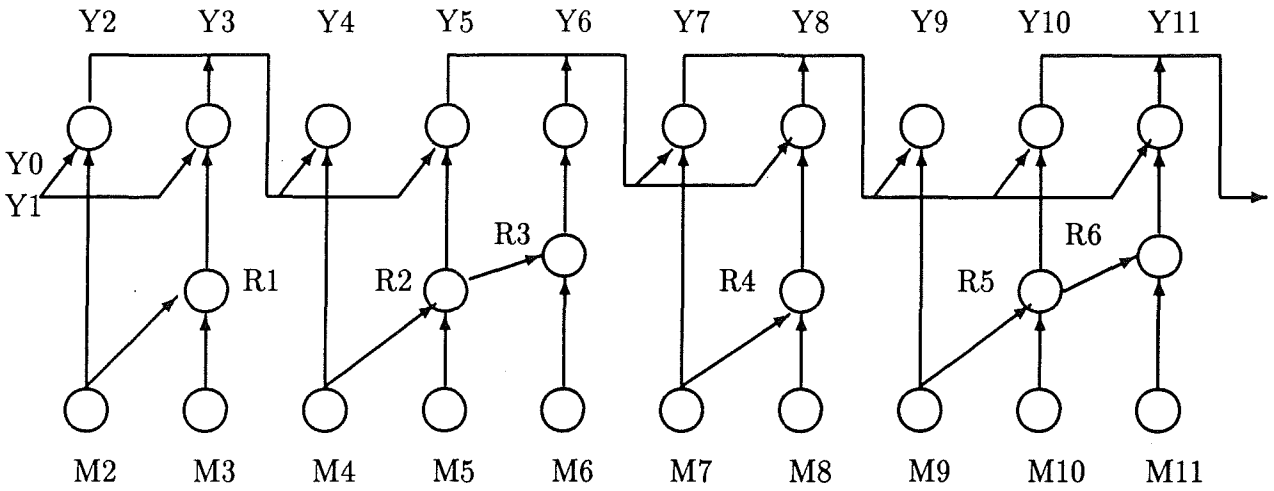

Figure 6: The computation tree in matrix multiplications for $p = 4$.

**procedure** construct_schedule_for_lr($p$, components)
1. call procedure build_computation_tree(components of all types);
2. call procedure FU_slot_allocation(tree, slot_sets);
3. transform the computation tree with allocated functional unit slots into a schedule;

**end** (construct_schedule_for_lr)

To illustrate the following phases, we assume $p = 4$ and choose $t_0 = 1$, thus having $w_0 = 0, w_1 = 2, w_2 = 1$. In the second phase, we construct a computation (dependence) tree for a period of the parallel schedule with those components obtained from the first phase, using the following procedure. The computation tree for $p = 4$ is shown in Figure 6.

**procedure** build_computation_tree(components of all types)
build redundant trees using the second type components $1, \ldots, w_1$;
**while** (there is a third type component)
**for** $i = w_1$ to 1 **do**
give a third type component to the $i$th redundant tree; **end for**
**end while**
**end** (build_computation_tree)

In the third phase, we allocate functional unit slots to operations in the computation tree. The functional unit slots are allocated to the final operations of the leading period and to the redundant operations of the succeeding period. The functional unit slots are allocated on a "most-urgent-first" policy, meaning that a functional unit slot is always assigned to compute the operation which if not computed by that slot would ruin the full functional unit utilization. Note that there are multiple ways of allocating the functional unit slots. We shall not detail the slot allocation procedure, which is similar to the one in [12], due to the space limit of this paper.

Figure 7 shows the computation tree for the first two periods expanded down into the arithmetic operation level with functional unit slot allocation. A box represents an arithmetic operation with operator inside the box. The number inside a box following the operator is the functional unit slot assigned to compute that operation at time indicated by the number, which will be the time step in

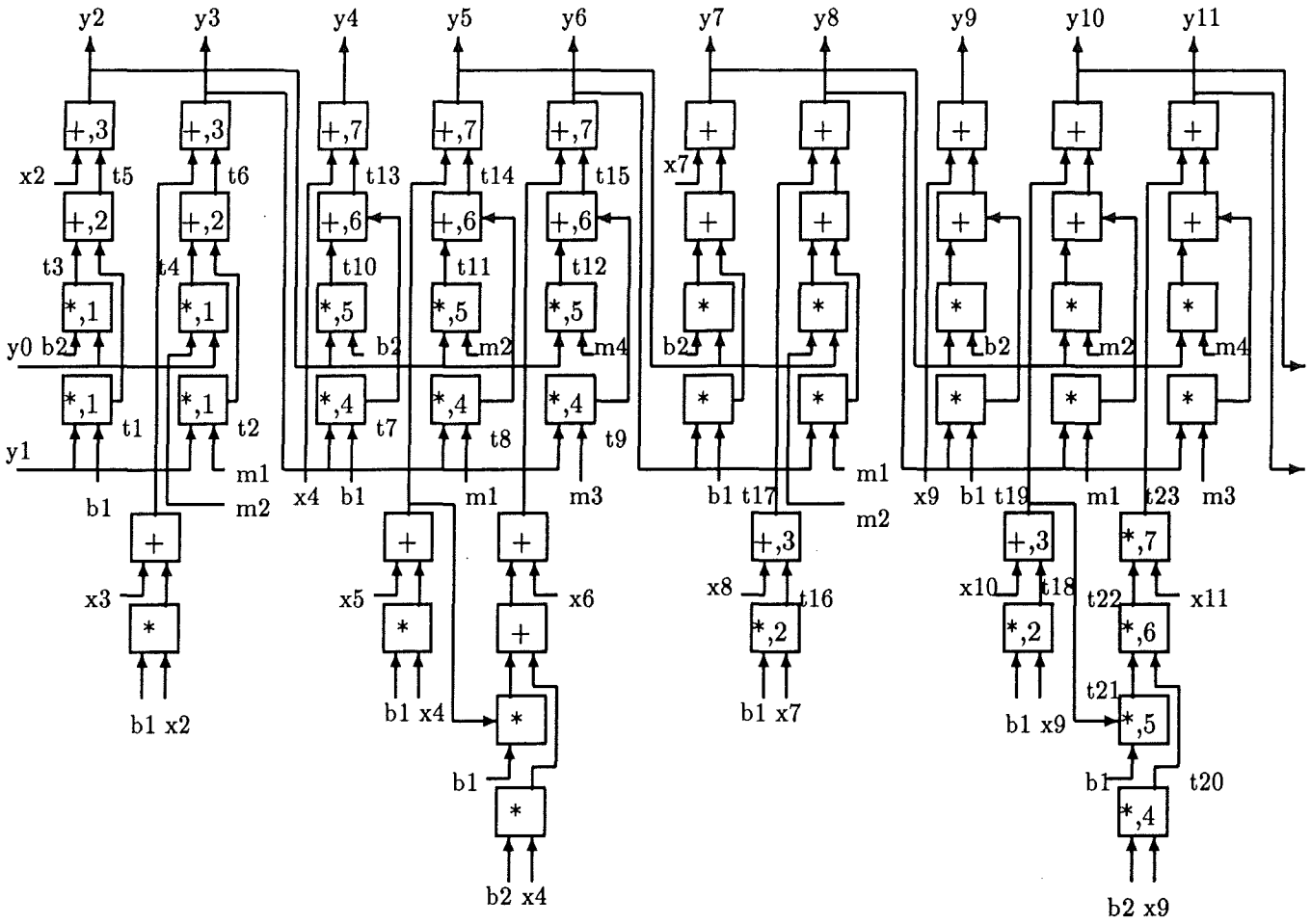


Figure 7: The computation tree in arithmetic operations with functional unit slot allocation for $p = 4$.

the loop body of the schedule. The operations below outputs $y_2$ through $y_6$ are in the leading period and those below outputs $y_7$ through $y_{11}$ in the succeeding period(the length of the period $\pi = 5$). The symbols $t_1$ through $t_{23}$ correspond to memory holding intermediate results. $b_1, b_2$ are the coefficients given in difference equation (2), $m_1 = b_1^2 + b_2, m_2 = b_1 b_2, m_3 = m_2 b_1 + b_1 b_2, m_4 = m_2 b_1 + b_2^2$ are precomputed coefficients, since these same coefficients are used throughout all the periods.

In the fourth phase, we write the computation tree with functional unit slot assignments into the final parallel schedule. We compose the final computation of the leading period and the redundant computation of the succeeding period into the loop body of our schedule, combined with the startup code, and express it into a parallel schedule as follows. In the schedule, $x[i]$ represents the $i$th sample input, $y[i]$ the $i$th sample output. Array $t[k][23]$ holds the intermediate results and its element $t[k][j]$ for $1 \geq j \geq 23$ is represented by $t_j$ in Figure 7. Obviously, the memory for intermediate results can be reduced to $t[2][23]$, but we choose not to use any storage optimization in favor of clarity. Before the program execution, the sample inputs $x[0], \ldots, x[6]$ are read in and stored. During each iteration, $\pi = 5$ sample inputs are read in and stored. This algorithm needs to store 10 sample inputs in each

iteration – resulting in a small hardware cost. Each step in the startup code is executed in parallel by $p = 4$ functional units.

1. $t[0][16] = b_1 * x[2]$, $t[0][18] = b_1 * x[4]$;
2. $t[0][17] = x[3] + t[0][16]$, $t[0][19] = x[5] + t[0][18]$;
3. $t[0][20] = b_2 * x[4]$, $t[0][21] = b_1 * t[0][19]$,
4. $t[0][22] = t[0][20] + t[0][21]$,
5. $t[0][23] = t[0][22] + x[6]$,

**for** $k = 0$ **to** *no more sample input* **do** *each step in parallel with p functional units*

1. $t[k+1][1] = b_1 * y[\pi k + 1]$, $t[k+1][2] = b_2 * y[\pi k]$,
   $t[k+1][3] = m_1 * y[\pi k + 1]$, $t[k+1][4] = m_2 * y[\pi k]$;
2. $t[k+1][5] = t[k+1][1] + t[k+1][2]$, $t[k+1][6] = t[k+1][3] + t[k+1][4]$,
   $t[k+1][16] = b_1 * x[\pi k + 7]$, $t[k+1][18] = b_1 * x[\pi k + 9]$;
3. $y[\pi k + 2] = x[\pi k + 2][1] + t[k+1][5]$, $y[\pi k + 3] = t[k][17] + t[k+1][6]$,
   $t[k+1][17] = x[\pi k + 8] + t[k+1][16]$, $t[k+1][19] = x[\pi k + 10] + t[k+1][18]$;
4. $t[k+1][7] = b_1 * y[\pi k + 3]$, $t[k+1][8] = b_2 * y[\pi k + 3]$,
   $t[k+1][9] = m_1 * y[\pi k + 3]$, $t[k+1][20] = b_2 * x[\pi k + 9]$;
5. $t[k+1][10] = b_2 * y[\pi k + 2]$, $t[k+1][11] = m_2 * y[\pi k + 2]$,
   $t[k+1][12] = m_4 * y[\pi k + 2]$, $t[k+1][21] = b_1 * t[k+1][19]$;
6. $t[k+1][13] = t[k+1][7] + t[k+1][10]$, $t[k+1][14] = t[k+1][8] + t[k+1][11]$,
   $t[k+1][15] = t[k+1][9] + t[k+1][12]$, $t[k+1][22] = t[k+1][20] + t[k+1][21]$;
7. $y[\pi k + 4] = x[\pi k + 4][1] + t[k+1][13]$, $y[\pi k + 5] = t[k][19] + t[k+1][14]$,
   $y[\pi k + 6] = t[k][23] + t[k+1][15]$, $t[k+1][23] = x[\pi k + 11] + t[k+1][22]$;

**end for**

Clearly, the loop body realized the full resource utilization, four to the redundant tree, four The schedule achieves an execution time for producing $N$ sample outputs ong chains $T_4 = 7/5N$, yielding a speedup of $20/7$ over sequential time $4N$, with a schedule size of 28 operations in the loop body, 7 operations for each functional unit. Each functional unit needs only a controller (i.e., a control table) of no more that 12 microinstructions (including the operations in the startup steps).

Table 1 gives the improvement in speedup over the sequential schedule by our schedule, generated using Harmonic Scheduling, on the previously published fastest parallel schedule[3], and the savings in the schedule size(i.e., the number of operations in the loop body of the schedule plus the startup) by our schedule from that schedule for the second order difference equation. The speedup improvement is determined by

$$\frac{(\textit{speedup by our schedule}) - (\textit{speedup by previously published fastest schedule})}{\textit{speedup by previously published fastest schedule}} 100.$$

The savings in control overhead is calculated by the reduction in the size of the schedule:

$$\frac{(\textit{size of previously published fastest schedule}) - (\textit{size of our schedule})}{\textit{size of previously published fastest schedule}} 100.$$

As the order of the difference equation increases, the speedup improvement and schedule size savings by our schedule increase for number of functional units $> 2$. Note that for some number of functional

| number of FU's | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|
| previous fastest schedule's $S_p$ | 2 | 32/13 | 50/17 | 24/7 | 98/25 | 128/29 | 54/11 | 200/37 |
| our schedule's $S_p$ | 12/5 | 20/7 | 10/3 | 42/11 | 56/13 | 24/5 | 90/17 | 110/19 |
| speedup improved by % | 20% | 16% | 13% | 11% | 9.9% | 8.7% | 7.8% | 7.1% |
| previous fastest schedule's size | 54 | 104 | 170 | 252 | 350 | 464 | 594 | 740 |
| our schedule size | 36 | 36 | 120 | 180 | 252 | 336 | 432 | 540 |
| schedule size saved by % | 33% | 65% | 29% | 29% | 28% | 28% | 27% | 27% |

Table 1: The improvement in speedup and the reduction in schedule size by our schedule on the previously published fastest schedule [3] for 2nd-order difference equation.

units such as $p = 4$, the schedule size can be made even smaller due to some properties [12] of the integer solutions to inequalities (4).

# 6 Summary and Remarks

Linear difference equations are fundamental equations that describe many important signal processing applications. In order to achieve good throughput for high sample rate digital filter applications, we need to effectively parallelize the classes of digital filters described with linear difference equations – typically a difficult task due to the recurrences in the data dependences of the linear difference equations. We presented a novel approach, *Harmonic Scheduling*, that exploits these recurrences and generates optimal schedules for parallel evaluation of linear difference equations with resource constraints. Furthermore, we can derive a parallel schedule which further minimizes the control overhead (microprograms needed to control the functional units), given an execution time with resource constraints. We described a Harmonic Scheduling algorithm for generating optimal schedules having minimum control overhead for digital filters described by second-order difference equations with resource constraints. The parallel schedules generated with Harmonic Scheduling achieve an execution time of

$$T_p = \frac{8p-4}{p(p+1)}N, p > 2,$$

yielding a speedup of $(p+1)/(2-1/p) \geq p/2$ over sequential schedules, which also have the least amount of control overhead. This is the fastest parallel schedule up to date for the second-order difference equation (2). It also has the smallest schedule size in comparison with the previously published parallel scheudels. Currently, we are working on relaxing some of the simplifying assumptions made in this paper to make the target architecture and the design model more realistic. In particular, we are working on adapting this technique to find optimal schedules for $m$th order difference equations with resource constraints under practical conditions such as using multiple-cycle operations (e.g., multiplication), and functional units of different types, operations and delays. Future work needs to address the effect of additional storage introduced by temporaries needed for redundant operations,

as well as the impact of connectivity and bindings on Harmonic Scheduling.

# References


[1] Chen, S. C., Kuck, D. and Sameh, A. H., "Practical Parallel Band Triangular System Solvers", ACM Transactions on Math. Software, Vol. 4, pp. 270-277, Sept, 1978.

[2] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and pipelined VLSI implementation of signal processing algorithms", pp. 257-276, *VLSI and modern signal processing*, S. Y. Kung, H. J. Whitehouse, T. Kailath, editors, Prentice-hall, 1985.

[3] D. Gajski, "An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines", IEEE Transactions on Computers, Vol. c-30, No.3, March 1981.

[4] G. Goossens, J. Vandewalle and H. De Man, "Loop optimizations in register-transfer scheduling for DSP systems", *Proceedings of the ACM/IEEE 26th Design Automation Conference*, 1989.

[5] R. Hartley and A. Casavant, "Tree height minimization in pipelined architectures", *Proceedings of 1989 IEEE International Conference on CAD*, Santa Clara, California, November 1989.

[6] C. T. Hwang, Y. C. Hsu and Y. L. Lin, "Scheduling for functional pipelining and loop winding", pp. 764-769, *Proceedings of the ACM/IEEE 28th Design Automation Conference*, San Francisco, California, June 1991.

[7] L. Hyafil, and H. T. Kung. "The Complexity of Parallel Evaluation of Linear Recurrence", JACM, Vol. 24, No.3, pp. 513-521, July 1977.

[8] Kogge, P. and Stone, H., "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", IEEE Transactions on Computer, Vol., C-22, No.8, August 1973.

[9] D. A. Lobo and B. M. Pangrle, "Redundant operation creation: a scheduling optimization technique", *Proceedings of the ACM/IEEE 28th Design Automation Conference*, pp. 775-778, San Francisco, California, June 1991.

[10] A. Nicolau and R. Potasman, "Incremental tree height reduction for high level synthesis", *Proceedings of the ACM/IEEE 28th Design Automation Conference*, pp. 770-774, San Francisco, California, June 1991.

[11] Nicolau, A., Wang, H. G., "Optimal Schedules for Parallel Prefix Computation with Bounded Resources", Proceeding of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming , Williamsburg, Virginia, April 21-24, 1991.

[12] A. Nicolau, H. Wang, "Speedup of band linear recurrence in the presence of resource constraints", Tech. Rep. 92-12, Information and Computer Science Department, University of California, Irvine, Dec. 1991.

[13] N. Park and A. C. Parker, "A software package for synthesis of pipelines from behavioral specifications", *IEEE Trans. CAD*, Vol. 7, No. 3, March 1988.

[14] T. W. Parks and C. S. Burrus, *Digital Filter Design*, Chapter 8, John Wiley & Sons, Inc., 1987.

[15] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based synthesis", *Proceedings of the ACM/IEEE 27th Design Automation Conference*, Orlando, Florida, June 1990.

[16] H. Trickey, "Flamel: a high-level hardware compiler", *IEEE Trans. on CAD*, Vol. CAD-6, No. 2, March 1987.

[17] P. P. Vaidyanathan, "Low-noise and low-sensitivity digital filters", *Handbook of Digital Signal Processing, Engineering Applications*, pp. 359-480, Academic Press, Inc., 1987.